

CEWES MSRC/PET TR/97-02

Some Performance Issues Associated with CEWES MSRC Scalable Architectures

by

Steve Bova

DoD HPC Modernization Program

Programming Environment and Training

CEWES MSRC



**Work funded by the DoD High Performance Computing
Modernization Program CEWES
Major Shared Resource Center through**

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

Views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of Defense Position, policy, or decision unless so designated by other official documentation.

Some Performance Issues Associated with CEWES MSRC Scalable Architectures

S.W. Bova*

December 5, 1997

Abstract

Some typical performance issues associated with sparse matrix codes which use the Message Passing Interface and distributed memory machines will be discussed. In particular, average, sustained, floating point performance will be examined as a function of problem size and number of processors for an unstructured fluid mechanics solver. These issues will be examined within the context of the iterative solution of nonsymmetric, linear systems of equations, such as those that arise from finite volume approximations to convection-diffusion problems. A brief description of the parallel CGSTAB (stabilized bi-conjugate gradient) iterative method will be followed by comparisons of its performance on the IBM SP, SGI Origin 2000, and Cray T3E at the Corps of Engineers Waterways Experiment Station Major Shared Resource Center (CEWES MSRC). In general, these machines have chips that are rated at several hundred Mflops, but actual performance obtained in practice is much less. Sustained performance is a strong function of cache and communication performance. The aim of this report is not to determine which machine is “better” for certain classes of problems, but rather to illustrate performance trade-offs that users can expect to address in most problems run on these machines.

1 Introduction

As part of the Department of Defense High Performance Computing Modernization Program, there has been a substantial increase in the computational resources available at the U.S. Army Corps of Engineers Waterways Experiment Station Major Shared Resource Center (CEWES MSRC). In particular, as part of the Performance Level 2 upgrade, an IBM SP, an SGI Origin 2000, and a Cray T3E have been installed (1). These machines represent quite a departure from the pre-existing Cray Y/MP and C90 vector processors. The most obvious difference in these new machines is in the use of commodity, superscalar RISC chips to produce highly parallel, distributed memory architectures. In general, these machines have chips that are rated at several hundred Mflops each, but actual performance obtained in practice is much less. Achievable sustained performance is a strong function of how effectively a given program can use the cache and the expense of interprocessor communication. The CEWES MSRC webpage (<http://apollo.wes.hpc.mil/hardware/systems.html>) gives the hardware data for the three machines listed in Table 1. The Mflops per processor were obtained simply by dividing the total Gflops in the table by the total number of processors.

This report describes some numerical experiments that were performed on the machines listed in Table 1. In the remaining sections, a model problem is described, followed by a brief description of the parallel methodology. Then parallel performance results are presented and discussed.

*CEWES On-site CFD Lead for PET, Mississippi State University

Machine	Number & Type of PE's	Total Memory	Total Gflops	R_p
O2000	(32) 200 MHz MIPS R10000	16 GB	11.5	360
T3E	(312) 450 MHz DEC Alpha	80 GB	281	900
SP	(256) 135 MHz IBM RS/6000	256 GB	138	540

Table 1: Specifications of CEWES MSRC Architectures. Performance rates are vendor-advertised peak performance. R_p is the performance rate in Mflops per processor.

2 The Model Problem

Selection of a model or benchmark problem is somewhat problematic, because there are two basic requirements which are in conflict: the problem should be representative of large-scale computational mechanics codes, and yet should also be small enough to run many times on several different problem sizes and machines in a reasonable amount of time. For this study, the model problem is a two-dimensional, nonlinear, steady-state, scalar, convection-diffusion-reaction problem of the form

$$\nabla \cdot \mathbf{F}(c) - \nabla \cdot [K(c) \nabla c] - R(c) = 0, \quad (1)$$

where c is the unknown concentration, $\mathbf{F}(c)$ is the nonlinear convective flux, $K(c)$ is the diffusivity, and $R(c)$ is the reaction term. The details of the parallel finite volume algorithm used to approximate this equation may be found in [1]. Equation (1) is linearized and discretized via an upwind finite volume method using unstructured triangulations. This approach leads to a sequence of linear systems of equations of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad (2)$$

where \mathbf{x} represents the unknown concentrations, \mathbf{b} is the forcing function which arises due to the reaction and boundary condition terms, and \mathbf{A} is a sparse, square, nonsymmetric coefficient matrix. It should be noted that the solution of systems of this form is a fundamental problem in computational mechanics. There are two main distinct ways to solve such systems, which can be broadly classified as either direct or iterative. Direct methods are based on Gaussian elimination. They are in general more robust but harder to implement and scale on distributed memory architectures. On the other hand, iterative methods are easier to parallelize, but are not as robust, particularly for nonsymmetric matrices.

3 Parallel Methodology

3.1 Communication Requirements

With finite element or finite volume methods, the parallelization of the matrix construction process is trivial. Parallel solution of the matrix system (2), however, is not. In this report, the CGSTAB iterative method is used [2]. This Krylov method also requires an effective preconditioner for efficiency. In this study, simple diagonal preconditioning is used exclusively and it should be noted that this may not be very effective for certain problems in this class. However, more complex preconditioners are harder to parallelize. The details of the CGSTAB algorithm are available in the literature and will not be repeated here. However, it is important to note that each iteration requires four dot products and 2 matrix-vector products. To parallelize a Krylov method using diagonal preconditioning, there are two essential steps which require communication: the matrix vector product requires point-to-point communication for exchange of data on the processor interfaces, and the dot products require global reduction operations.

In traditional finite element and finite volume schemes (*e.g.* see [3]), the construction of the global matrix \mathbf{A} is performed by looping over the individual cells and calculating local element or edge contributions \mathbf{A}_e , which are then accumulated, or summed, to obtain the global coefficient matrix \mathbf{A} ,

$$\mathbf{A} = \sum_{e=1}^E \mathbf{A}_e. \quad (3)$$

The element contributions \mathbf{A}_e are dense, but small. For example, if (1) is discretized using linear triangles, \mathbf{A}_e is a dense 3×3 matrix for each element. Typically, some sparse storage scheme is used to store the assembled matrix \mathbf{A} , and the matrix vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ can be computed using a corresponding indirect addressing scheme. An alternative, “element-by-element” approach for computing the matrix-vector product is to store only the element matrices \mathbf{A}_e and never assemble the global matrix. Instead, the effect of the assembly process is achieved by summing the local element contributions to the matrix-vector products, *viz.*

$$\mathbf{y} = \sum_{e=1}^E \mathbf{A}_e \mathbf{x} \quad (4)$$

This approach leads to a very natural sparse storage scheme which is easily done in parallel [4].

The basic parallel strategy is implemented in the present performance study to first partition the triangles in the grid using the METIS package (<http://www.cs.umn.edu/~karypis/metis/metis.html>). Note that this implies that grid points are shared. Next, form and store the local element matrices \mathbf{A}_e and the vector \mathbf{b} . Then the essence of the parallel Krylov method involves the computation of the matrix vector products in parallel, and the exchange of the interface values using message-passing sends and receives. The MPI library was used to make measurements for this report. Specifically, the non-blocking routines `MPI_Isend()` & `MPI_Irecv()` were used for the point-to-point communication. Each processor’s contribution to the dot product is calculated in parallel and then combined via a call to `MPI_Allreduce()` to accomplish the reduction operation.

3.2 Message Passing Bandwidth

To gain insight into the message-passing performance of the above algorithm, the MPI bandwidth of each of the three machines in Table 1 was determined experimentally. This was done by sending a message of a given size one-way between two processors. This experiment was repeated one hundred times for each message length and the results averaged to obtain the results shown in Figure 1. First notice that there are two curves shown for the IBM SP. On this machine, there are two message passing subsystems available to the user: the internet protocol (IP) and the user space (US). If the US is used on a processor, then no other process may access the high-speed switch. However, the performance advantages of the US are substantial. It is interesting to note that the T3E and the Origin 2000 have approximately the same bandwidth. The US subsystem on the SP consistently gives about half the bandwidth of the other two machines. The IP subsystem consistently gives about an order of magnitude less bandwidth than the US.

It is also instructive to consider the transit time of a message. To this end, the bandwidth data were reduced to obtain the plot shown in Figure 2. Note that the transit times are dominated by latency until the message length is about 1 kbyte. The approximate values of the experimentally obtained MPI latencies on each of the three machines are taken from the figure and given in Table 2. (Here, latency is obtained by simply taking the minimum transit time.) For Krylov algorithms such as CGSTAB, the latencies are particularly important

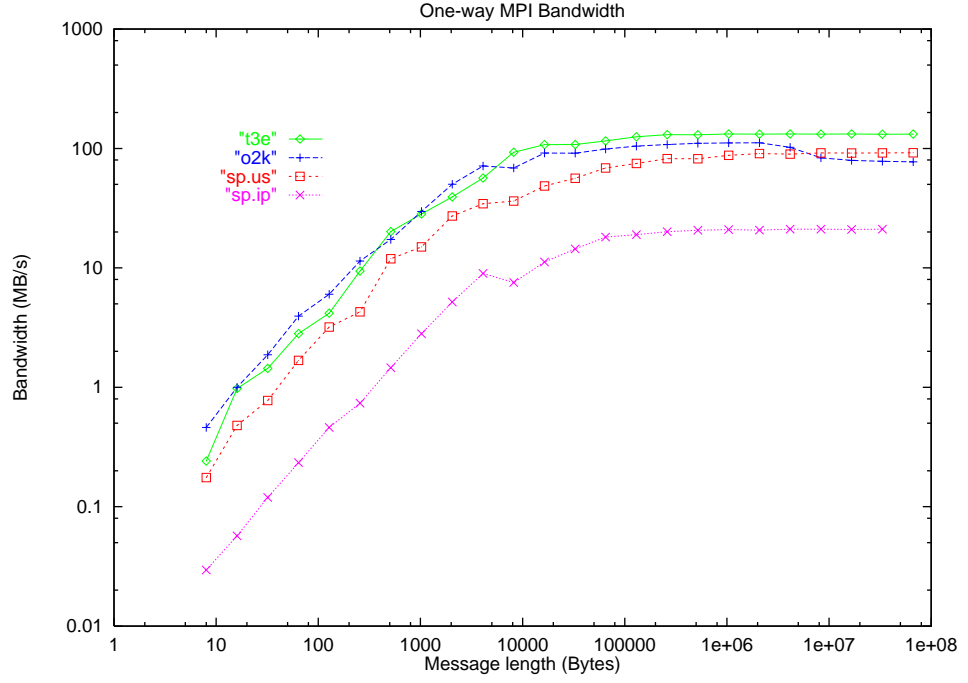


Figure 1: One-way MPI bandwidth as a function of message length.

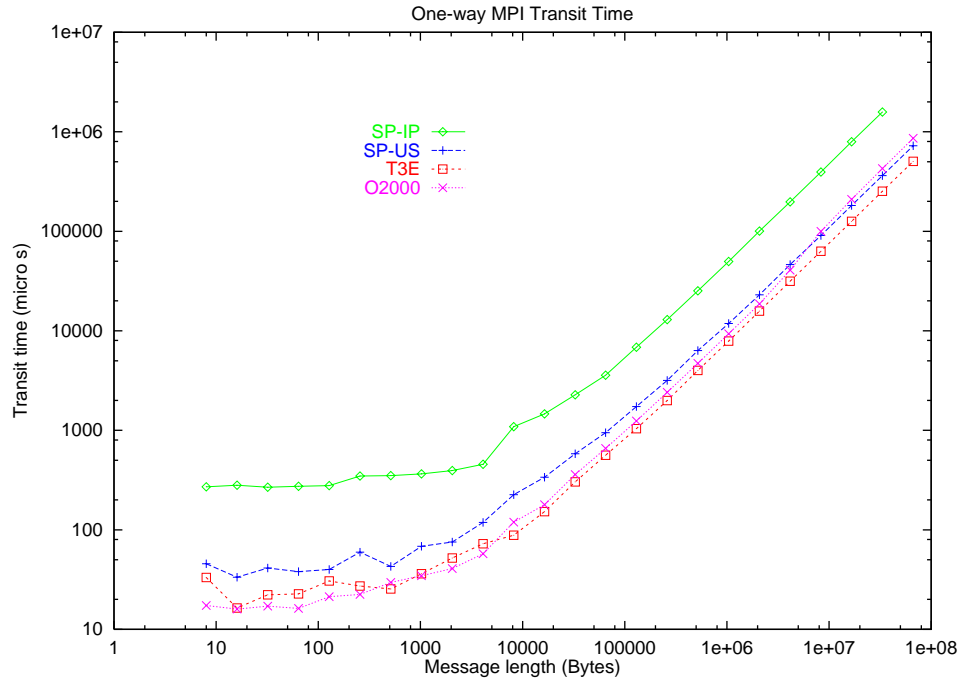


Figure 2: One-way MPI message transit time as a function of message length.

Machine	Latency (μs)
SP (IP)	300
SP (US)	30
T3E	16
O2000	16

Table 2: Approximate measured MPI latencies.

because of the inner product calculation: the latency suggests a lower bound on the time required to perform a global reduction operation to a single 64-bit number. This type of global (as opposed to point-to-point) communication cannot be overlapped under the current MPI standard.

In order to estimate typical message lengths encountered by a finite volume code when exchanging data on processor interfaces, consider the following. In two dimensions, assume that there are 10^5 grid points distributed among 100 processors. This leads to 10^3 grid points per processor, which is approximately a square of 32 grid points per side. So each exchange would involve lines of 32 grid points. If it is further assumed that there are five degrees of freedom per grid point, and if each word is 64 bits, the message length would be approximately 1kB. In three dimensions, assume that there are 10^6 grid points distributed among the same 100 processors. This amounts to a cube of approximately 21^2 grid points per side. Assuming six degrees of freedom per grid point, the message length to exchange a single plane of data would be approximately 22 kbytes. These estimates indicate that in three dimensions, the maximum bandwidth shown in Figures 1 and 2 can be readily obtained, but probably not in two dimensions.

4 Numerical Experiments

In order to illustrate some basic performance issues, the model problem was run on the three machines for a given set of boundary conditions using problem sizes of 7,800, 19,000, 43,000, and 93,000 triangles. For example, Figure 3 shows the 7,800 element grid as partitioned by METIS for computation on four processors.

The average sustained performance is examined as a function of problem size and the number of processors. The effect of two simple optimizations is also considered. First, recall that the finite volume grid is unstructured. Hence a front width minimization ordering is performed on the grid points in order to improve cache performance (*e.g.* see [5]). To see why reordering should improve cache performance, consider that for unstructured grids, the grid ordering is generally not such that consecutively labeled points and elements are adjacent to one another in the grid. For example, if the elements are randomly ordered, then it is not likely that consecutively labeled elements will update grid points that have already been loaded into cache. A front width minimization numbering strategy consecutively labels the elements which share grid points. Hence, as the elements are processed, all contributions to a single grid point can probably be calculated while the corresponding array element is in cache. Second, the triangles which support grid points on the processor interfaces are ordered first in the list so that the communication to perform the exchange for the matrix vector products can be overlapped with useful floating point calculations. This can be achieved by breaking the loop for the matrix-vector product into two parts. The first part loops over the first T_i interface triangles and computes their contributions as in (4). Next the nonblocking sends and receives are initiated. Then the second part calculates the contributions from the interior triangles labeled $e = T_i$ to E . Finally, the communication operation is completed

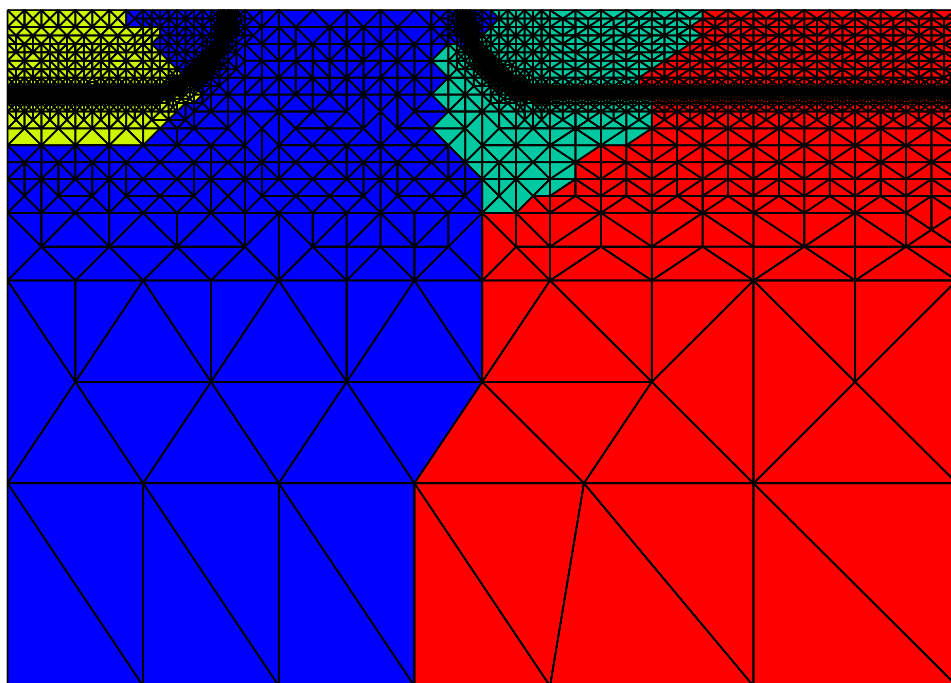


Figure 3: Example mesh of 7,800 elements as partitioned by METIS for four processors.

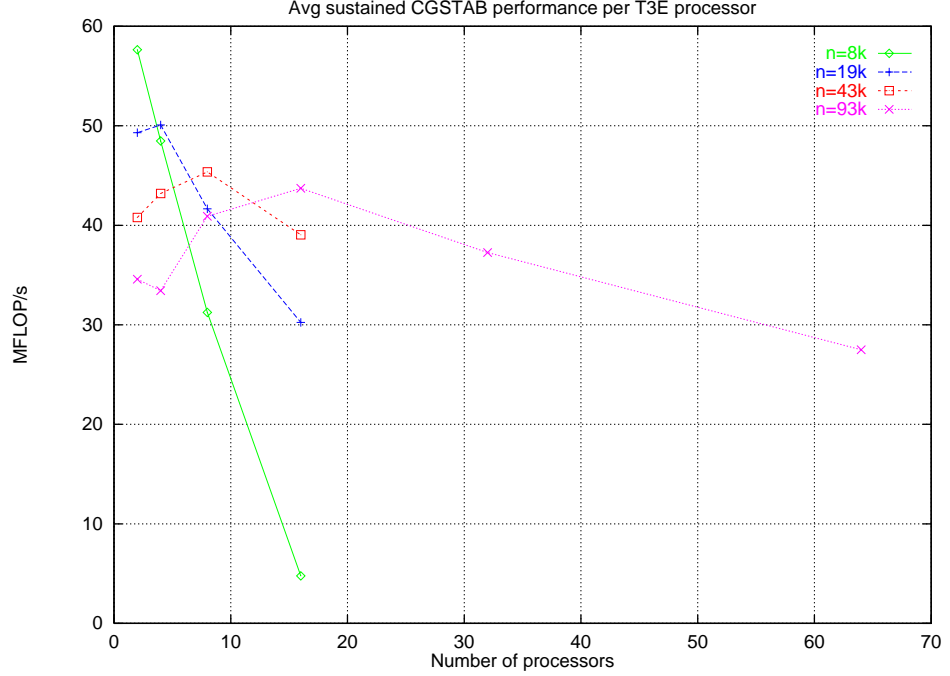


Figure 4: Average sustained CGSTAB performance per T3E processor as a function of the number of processors and parameterized by problem size.

via a call to `MPI_Wait()` and the off-processor contributions accumulated.

All timings were obtained with `MPI_Wtime()`, so that they represent elapsed wall clock time. These results are still reproducible, however, because on the SP (with the US protocol) and the T3E a user exclusively owns a processor once it has been allocated. This is not the case on the Origin 2000, which is like a traditional time-sharing Unix system. Great care was taken in the studies described here to ensure that no other users were sharing processors during the O2000 experiments. These are exclusive of I/O, but represent the average of a single CGSTAB solve over the course of obtaining the nonlinear solution.

The flop counts were obtained by hand; addition, subtraction, multiplication, and division operations were each counted as a single flop. In general, the Mflop rate is not a good metric for benchmark evaluations because of the possible ambiguity in its definition. The focus of this report is to illustrate certain performance issues, not to establish a benchmark standard to be run by many different users on many different machines over an extended period of time. In this case, the Mflop rate is an interesting metric (if counted consistently) because it gives an estimate of the fraction of the peak floating point performance that can be achieved in practice. Furthermore, since the flop counts are consistent and the data sets are the same across the machines, a Mflop rate can be interpreted as inverse time.

First, consider the average, sustained *per processor* performance of the CGSTAB algorithm. Figures 4-6 present the Mflop rates as a function of the number of processors and parameterized by problem size for each of the three machines. This data was obtained with both the front width minimization ordering and the overlapped exchange.

First, note that the general trends on all three machines are the same. For example, consider the effect of problem size on floating point performance. For two processors, there is relatively little communication; the calculation is CPU bound. In this case, floating

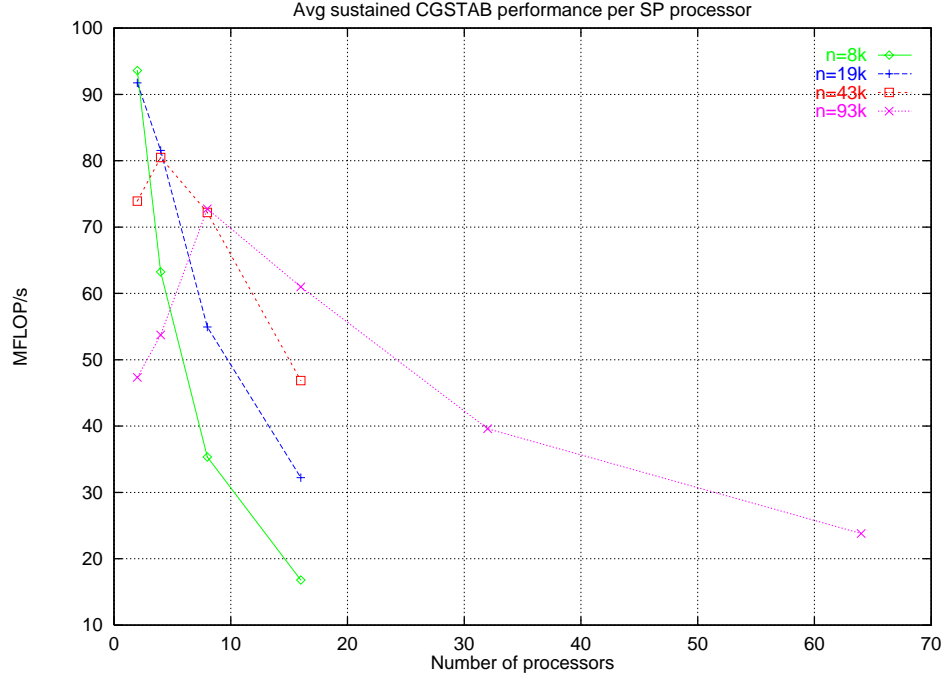


Figure 5: Average sustained CGSTAB performance per IBM SP processor as a function of the number of processors and parameterized by problem size.

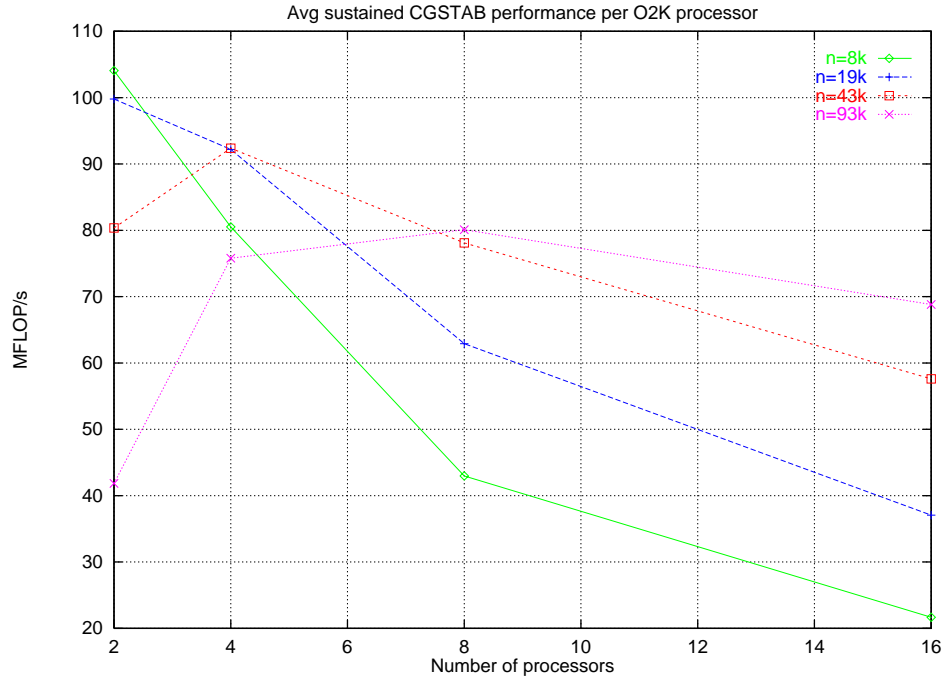


Figure 6: Average sustained CGSTAB performance per processor as a function of the number of O2000 processors and parameterized by problem size.

point performance decreases with increasing problem size. The explanation is that for small problem sizes, the data fits into cache and high performance is obtained. As the problem size is increased, the data becomes too large to fit into cache and floating point performance can decrease by as much as a factor of three. On the other hand, for sixteen processors there is substantially more communication than in the two processor case. Here the observation is reversed: floating point performance increases with increasing problem size. In this case, for small problem sizes the calculation is communication bound because there simply isn't enough work to offset the cost of communication. Increasing the problem size provides more work to offset the communication overhead, thereby increasing floating point performance. Also note that on each machine for the moderate to large problem sizes there is an optimum number of processors at which the maximum floating point performance is obtained. Finally, note that the highest per-processor performance of the T3E on the largest problem size is approximately half that of the other two machines. However, the floating point performance on 64 T3E processors exceeds that which is obtained on 64 SP processors.

Next, the effect of the element reordering and communication overlap is considered. Figures 7-9 show the total floating point performance as a function of processor number for the 93,000 element case for the T3E, the SP, and the O2000, respectively. For the T3E, the optimizations are quite effective, especially the overlapped exchange. Note that for the SP, essentially no benefit is observed with these optimizations for smaller processor numbers. On the O2000, the optimizations have very little effect. These results are also somewhat counterintuitive; the non-overlapped slightly beats the overlapped case for small numbers of processors. This can perhaps be explained by a software pipeline effect. The overlapping requires that the loop for the matrix vector product be broken into two shorter loops, thus incurring additional overhead and breaking up the pipelining effect associated with the single, larger loop.

Finally, the best performing optimization for each of the three machines is shown in Figure 10 for the 93,000 element case. Surprisingly, on two processors, the floating point performance of this algorithm on the three machines is essentially identical. As the number of processors is increased, the floating point performance is consistently better on the O2000 and the SP than on the T3E. However, ultimately the performance on the T3E exceeds that of the other two machines. For the O2000 this happens simply because the available system was limited to 32 processors. The reasons are more interesting for the SP. On the T3E, communication costs are about half that of the SP (see Table 2), while at the same time the per processor performance of the CGSTAB algorithm is substantially slower than the SP. Hence, the relative cost of communication to computation is much lower on the T3E and the algorithm is more scalable.

5 Discussion

In general, the simple experiments described in this report show that the floating point performance on microprocessor-based scalable computing systems depends strongly on the problem size. The relative balance of cache and communication performance is particularly important for single processor efficiency and overall scalability. For the CGSTAB algorithm, the dot products can become very expensive to calculate if the problem size is not scaled with the number of processors. This is because for short messages, the message latency dominates. To calculate a dot product, there is first a reduction operation that gathers each processor's contribution to the sum to a single process. Then the result is broadcast to all the other processors. Hence the cost of this calculation grows with the number of processors. This is in contrast to the point-to-point exchange operation, which is local and tends to vary very little for reasonably partitioned grids. The cost of the global reduction

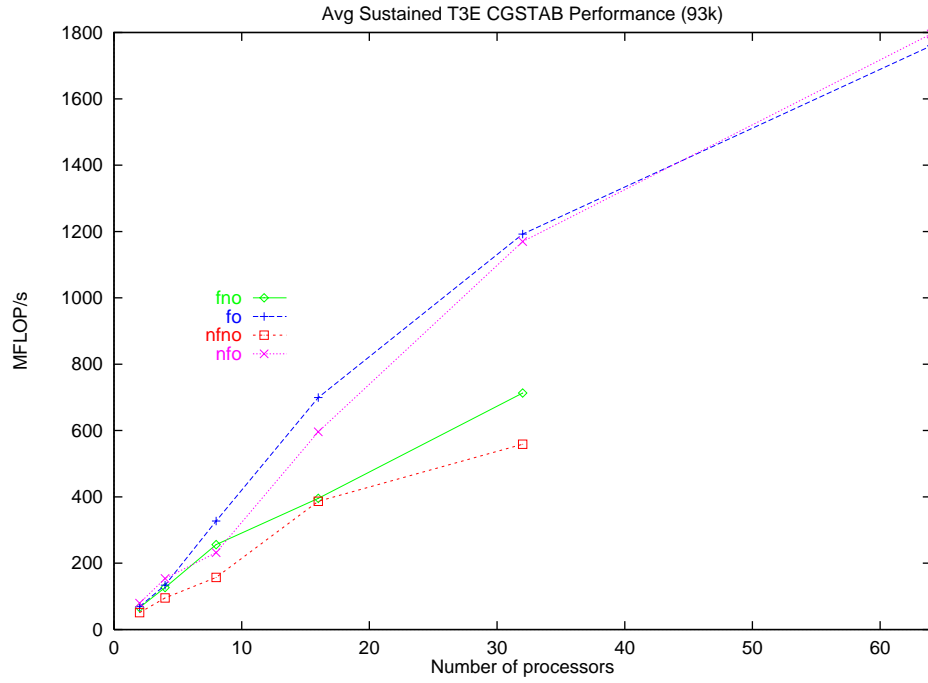


Figure 7: Effect of frontal ordering and communication overlap on T3E performance for the 93,000 element case. fno=frontal ordering, no overlap; fo=frontal ordering, with overlap; nfno=no frontal ordering or overlap; nfo=no frontal ordering but with overlap.

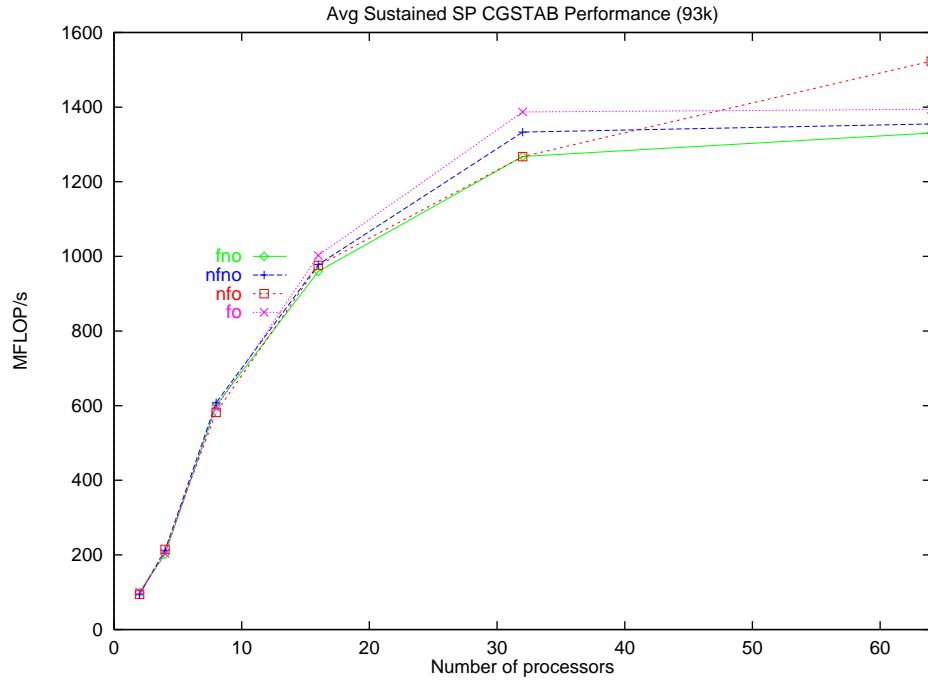


Figure 8: Effect of frontal ordering and communication overlap on SP performance for the 93,000 element case. fno=frontal ordering, no overlap; nfno=no frontal ordering or overlap; nfo=no frontal ordering but with overlap; fo=frontal ordering, with overlap.

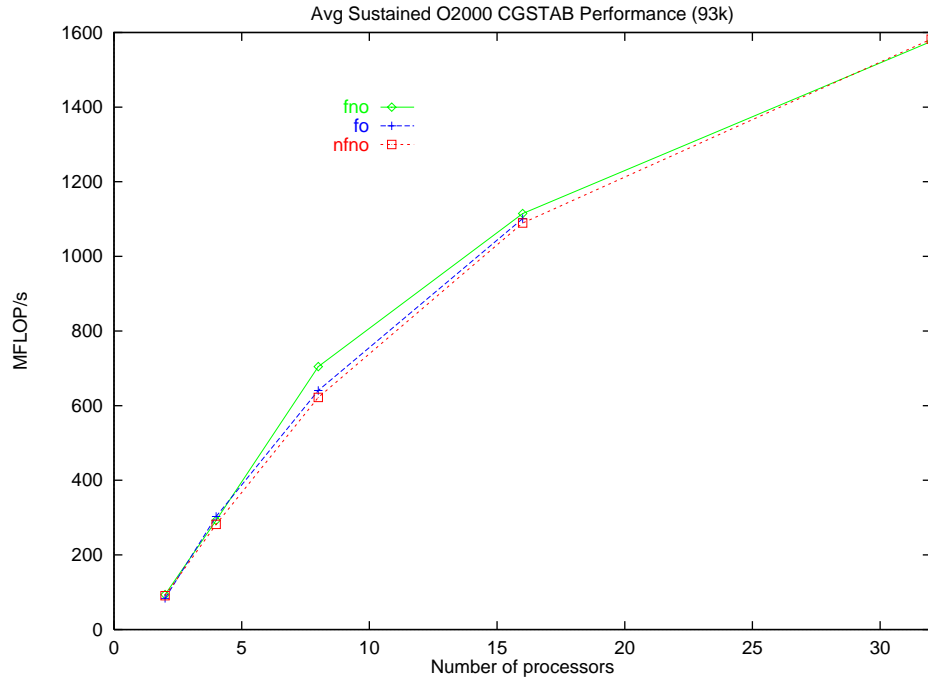


Figure 9: Effect of frontal ordering and communication overlap on O2000 performance for the 93,000 element case. fno=frontal ordering, no overlap; fo=frontal ordering, with overlap; nfno=no frontal ordering or overlap.

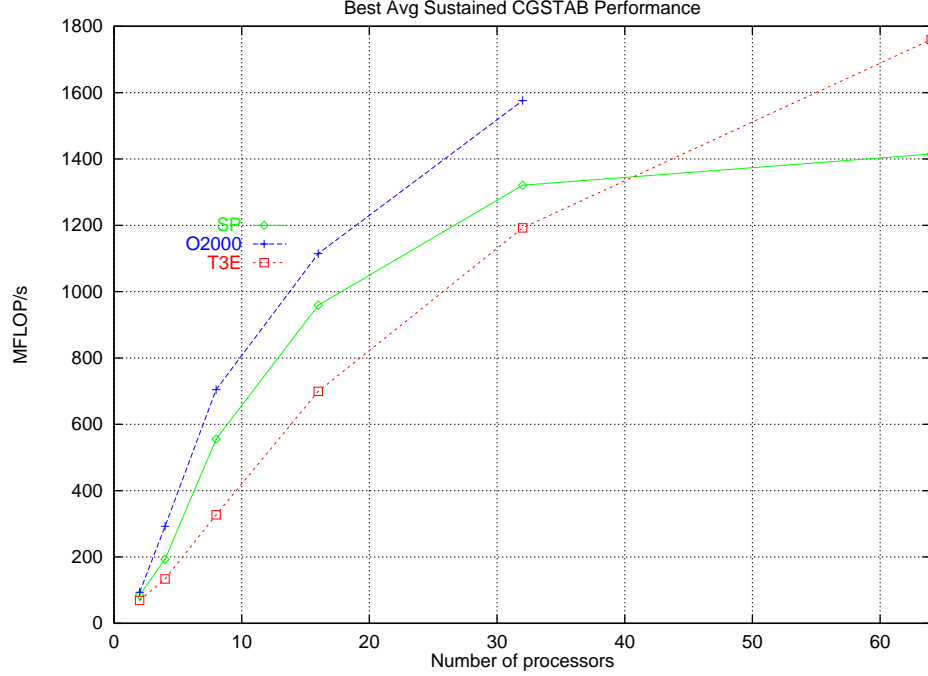


Figure 10: Comparison of performance for the 93,000 element case on the SP, T3E, and O2000.

operation is aggravated on slow networks and would be particularly bad on *e.g.* a network of workstations connect via ethernet.

In general, compared to the peak theoretical performance figures in Table 1 the realized performance of the algorithm all three machines is quite poor, ranging from at best about 30% to at worst about 1% of “peak”. On average, the CGSTAB code is running at about 10% of single processor efficiency on the T3E, 22% on the Origin 2000, and 15% on the SP. These numbers are representative of the performance of sparse matrix codes that are written in straightforward fortran on microprocessors. The importance of data locality cannot be overstated. If extensive manual blocking for cache performance, loop unrolling, and interleaving is performed, the efficiency can probably be increased to around 40-50%. But this leads to a compromise in code readability, portability and maintainability. If a production code is of substantial maturity and importance, then it may be well worth the optimization effort required.

Acknowledgement

This work was supported in part by a grant of HPC time from the DoD HPC Modernization Program.

References

- [1] S.W. Bova and G.F. Carey, “A distributed memory element-by-element scheme for semiconductor device simulation”, in preparation, 1997.

- [2] H.A. van der Vorst, “BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems”, *SIAM J. Sci. Stat. Comput.*, **13**, pp. 631-644, 1992.
- [3] E.B. Becker, G.F. Carey, and J.T. Oden, *Finite Elements: An Introduction*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
- [4] G.F. Carey, E. Barragy, R. McLay, and M. Sharma, “Element-by-element vector and parallel computations”, *Communications in Applied Numerical Methods*, **4**, pp. 299-307, 1988.
- [5] R. Löhner, “Renumbering strategies for unstructured-grid solvers on shared-memory, cache-based parallel machines”, AIAA paper 97-2045, presented at the Thirteenth AIAA Computational Fluid Dynamics Conference, Snowmass Village, CO, 1997.